Programming Languages and Compilers (CS 421)

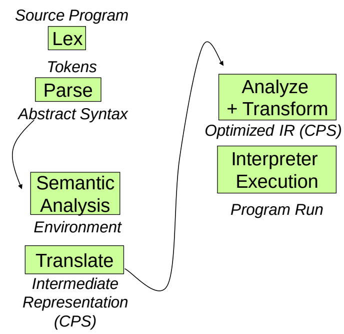
Sasa Misailovic

4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2017/CS421A

Based on slides by Elsa Gunter, which were inspired by earlier slides by Mattox Beckman, Vikram Adve, and Gul Agha

Major Phases of a PicoML Interpreter



Semantics

- Expresses the **meaning of syntax**
- Static semantics
  - Meaning based only on the form of the expression without executing it
  - Usually restricted to type checking / type inference

Dynamic semantics

- Method of **describing meaning of executing** a program
- Several different types:
  - Operational Semantics
  - Axiomatic Semantics
  - Denotational Semantics
- Different languages better suited to different types of semantics
- Different types of semantics serve different purposes

Operational Semantics

- Start with a simple notion of machine
- Describe how to execute (implement) programs of language on virtual machine, by describing how to execute each program statement (ie, following the *structure* of the program)
- Meaning of program is how its execution changes the state of the machine
- Useful as basis for implementations

Denotational Semantics

- Construct a function $\mathcal{M}$ assigning a mathematical meaning to each program construct
- Lambda calculus often used as the range of the meaning function
- Meaning function is compositional: meaning of construct built from meaning of parts
- Useful for proving properties of programs

Axiomatic Semantics

- Also called Floyd-Hoare Logic
- Based on formal logic (first order predicate calculus)
- Axiomatic Semantics is a logical system built from *axioms* and *inference rules*
- Mainly suited to simple imperative programming languages

Axiomatic Semantics

- Used to formally prove a property (*post-condition*) of the *state* (the values of the program variables) after the execution of program, assuming another property (*pre-condition*) of the state before execution
- Written :
  {**Precondition**} ***Program*** {**Postcondition**}

Natural Semantics ("Big-step Semantics")

- Aka Structural Operational Semantics, aka "Big Step Semantics"
- Provide value for a program by rules and derivations, similar to type derivations
- Rule conclusions look like

$$(C, m) \Downarrow m'$$

"Evaluating a command C in the state m results in the new state m' "

or

$$(E, m) \Downarrow v$$

"Evaluating an expression E in the state m results in the value v"

Simple Imperative Programming Language

- I $\in$ Identifiers
- N $\in$ Numerals
- B ::= true | false
  | B & B | B or B | not B | E < E | E = E
- E::= N | I | E + E | E * E | E - E | - E
- C::= skip | C;C | I ::= E
  | if B then C else C fi | while B do C od

Natural Semantics of Atomic Expressions

- Identifiers: $(k,m) \Downarrow m(k)$
- Numerals are values: $(N,m) \Downarrow N$
- Booleans: $(\text{true},m) \Downarrow \text{true}$
  $(\text{false},m) \Downarrow \text{false}$

Booleans:

$$\frac{(B, m) \Downarrow \text{false}}{(B \,\&\, B', m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{true} \quad (B', m) \Downarrow b}{(B \,\&\, B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(B \text{ or } B', m) \Downarrow \text{true}} \qquad \frac{(B, m) \Downarrow \text{false} \quad (B', m) \Downarrow b}{(B \text{ or } B', m) \Downarrow b}$$

$$\frac{(B, m) \Downarrow \text{true}}{(\text{not } B, m) \Downarrow \text{false}} \qquad \frac{(B, m) \Downarrow \text{false}}{(\text{not } B, m) \Downarrow \text{true}}$$

Relations

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \sim V = b}{(E \sim E', m) \Downarrow b}$$

- By U ~ V = b, we mean does (the meaning of) the relation ~ hold on the meaning of U and V
- May be specified by a mathematical expression/equation or rules matching U and V

Arithmetic Expressions

$$\frac{(E, m) \Downarrow U \quad (E', m) \Downarrow V \quad U \text{ op } V = N}{(E \text{ op } E', m) \Downarrow N}$$

where N is the specified value for U op V

Commands

Skip: $(\text{skip}, m) \Downarrow m$

Assignment: $$\frac{(E,m) \Downarrow V}{(k := E,m) \Downarrow m\,[k \text{ <-- } V\,]}$$

Sequencing: $$\frac{(C,m) \Downarrow m' \quad (C',m') \Downarrow m''}{(C;\, C', m) \Downarrow m''}$$

If Then Else Command

$$\frac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

$$\frac{(B,m) \Downarrow \text{false} \quad (C',m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$$

Example: If Then Else Rule

$$\frac{}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \Downarrow\ ?}$$

Example: If Then Else Rule

$$\frac{\overline{(x > 5, \{x \text{ -> } 7\}) \Downarrow ?}}{(\text{if } x > 5 \text{ then } y{:=}\ 2 + 3 \text{ else } y{:=}3 + 4 \text{ fi}, \{x \text{ -> } 7\}) \Downarrow\ ?}$$

Example: Arith Relation

```
        ? > ? = ?
   (x,{x->7})⇓?   (5,{x->7})⇓?
-----------------------------
      (x > 5, {x -> 7})⇓?
----------------------------------------------------
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7})
                     ⇓ ?
```


Example: Identifier(s)

```
        7 > 5 = true
   (x,{x->7})⇓7   (5,{x->7})⇓5
-----------------------------
      (x > 5, {x -> 7})⇓?
----------------------------------------------------
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7})
                     ⇓ ?
```


Example: Arith Relation

```
        7 > 5 = true
   (x,{x->7})⇓7   (5,{x->7})⇓5
-----------------------------
      (x > 5, {x -> 7})⇓true
----------------------------------------------------
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,  {x -> 7})
                     ⇓ ?
```


Example: If Then Else Rule

```
        7 > 5 = true                 --------------------
   (x,{x->7})⇓7   (5,{x->7})⇓5       (y:= 2 + 3, {x-> 7}
-----------------------------        ⇓ ?               .
      (x > 5, {x -> 7})⇓true
----------------------------------------------------
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7})
                     ⇓ ?
```


Example: Assignment

```
                              --------------------
        7 > 5 = true             (2+3, {x->7})⇓?
                              --------------------
   (x,{x->7})⇓7   (5,{x->7})⇓5   (y:= 2 + 3, {x-> 7}
-----------------------------
      (x > 5, {x -> 7})⇓true     ⇓ ?
----------------------------------------------------
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7})
                     ⇓ ?
```


Example: Arith Op

```
                                       ? + ? = ?
                                 (2,{x->7})⇓?   (3,{x->7}) ⇓?
                                 ----------------------------
        7 > 5 = true                 (2+3, {x->7})⇓?
                                 --------------------
   (x,{x->7})⇓7   (5,{x->7})⇓5       (y:= 2 + 3, {x-> 7}
-----------------------------
      (x > 5, {x -> 7})⇓true         ⇓?               .
----------------------------------------------------
(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7})
                     ⇓ ?
```

Example: Numerals

2 + 3 = 5

(2,{x->7})⇓2   (3,{x->7}) ⇓3

7 > 5 = true   (2+3, {x->7})⇓?

(x,{x->7})⇓7   (5,{x->7})⇓5   (y:= 2 + 3, {x-> 7}

(x > 5, {x -> 7})⇓true   ⇓ ?

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7})

⇓ ?

Example: Arith Op

2 + 3 = 5

(2,{x->7})⇓2   (3,{x->7}) ⇓3

7 > 5 = true   (2+3, {x->7})⇓5

(x,{x->7})⇓7   (5,{x->7})⇓5   (y:= 2 + 3, {x-> 7}

(x > 5, {x -> 7})⇓true   ⇓?

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7})

⇓ ?

Example: Assignment

2 + 3 = 5

(2,{x->7})⇓2   (3,{x->7}) ⇓3

7 > 5 = true   (2+3, {x->7})⇓5

(x,{x->7})⇓7   (5,{x->7})⇓5   (y:= 2 + 3, {x-> 7}

(x > 5, {x -> 7})⇓true   ⇓ {x->7, y->5}

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7})

⇓?

Example: If Then Else Rule

2 + 3 = 5

(2,{x->7})⇓2   (3,{x->7}) ⇓3

7 > 5 = true   (2+3, {x->7})⇓5

(x,{x->7})⇓7   (5,{x->7})⇓5   (y:= 2 + 3, {x-> 7}

(x > 5, {x -> 7})⇓true   ⇓ {x->7, y->5}

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7})

⇓ **{x->7, y->5}**

While Command

(B, m) ⇓ false

(while B do C od, m) ⇓ m

(1) (B,m) ⇓ true   (2) (C,m) ⇓ m'   (3) (while B do C od, m') ⇓ m''

(while B do C od, m) ⇓ m''

Example: While Rule

(x > 5, {x->2}) ⇓ false

(1) (x > 5, {x->7}) ⇓ true   (3) while x > 5 do x := x-5 od;

(2) (x := x-5, {x->7}) ⇓ {x->2}   {x -> 2}) ⇓ {x->2}

(while x > 5 do x := x-5 od, {x -> 7}) ⇓ {x->2}

While Command

$$\frac{(B, m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

$$\frac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m''}$$

The rule assumes the loop terminates!

While Command

$$\frac{(B, m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$$

$$\frac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m''}$$

The rule assumes the loop terminates!

$$\frac{\mathbf{???}}{\text{while } (x>0) \text{ do } x:=x+1 \text{od}, \{x\text{->}1\} \Downarrow \mathbf{???}}$$

Let's Try Adding Let in Command…

$$\frac{(E,m) \Downarrow v \quad (C,m[k\text{<-}v]) \Downarrow m'}{(\text{let } k = E \text{ in } C, m) \Downarrow m''}$$

Where

m''(y) = m'(y) for y ≠ k and
<u>if m(k) is defined</u>, m''(k) = m(k) or
<u>otherwise</u> m''(k) is undefined

Example

$$\frac{(5,\{x\text{->}17\}) \Downarrow 5 \quad \dfrac{\dfrac{(x,\{x\text{->}5\}) \Downarrow 5 \quad (3,\{x\text{->}5\}) \Downarrow 3}{(x+3,\{x\text{->}5\}) \Downarrow 8}}{(x:=x+3,\{x\text{->}5\}) \Downarrow \{x\text{->}8\}}}{(\text{let } x = 5 \text{ in } (x:=x+3), \{x \text{->} 17\}) \Downarrow ?}$$

Example

$$\frac{(5,\{x\text{->}17\}) \Downarrow 5 \quad \dfrac{\dfrac{(x,\{x\text{->}5\}) \Downarrow 5 \quad (3,\{x\text{->}5\}) \Downarrow 3}{(x+3,\{x\text{->}5\}) \Downarrow 8}}{(x:=x+3,\{x\text{->}5\}) \Downarrow \{x\text{->}8\}}}{(\text{let } x = 5 \text{ in } (x:=x+3), \{x \text{->} 17\}) \Downarrow \{x\text{->}17\}}$$

Recall: Where m''(y) = m'(y) for y ≠ k and m''(k) = m(k) if m(k) is defined, and m''(k) is undefined otherwise

Comment on Language Design

- Simple Imperative Programming Language introduces variables ***implicitly*** through assignment
- The let-in command introduces scoped variables ***explictly***
- Clash of constructs apparent in awkward semantics – a question for language designers!

Interpretation Versus Compilation

- A **compiler** from language L1 to language L2 is a program that takes an L1 program and for each piece of code in L1 generates a piece of code in L2 of same meaning
- An **interpreter** of L1 in L2 is an L2 program that executes the meaning of a given L1 program
- Compiler would examine the body of a loop once; an interpreter would examine it every time the loop was executed

Interpreter

- An *Interpreter* represents the operational semantics of a language L1 (source language) in the language of implementation L2 (target language)
- Built incrementally
  - Start with literals
  - Variables
  - Primitive operations
  - Evaluation of expressions
  - Evaluation of commands/declarations

Interpreter

- Takes abstract syntax trees as input
  - In simple cases could be just strings
- One procedure for each syntactic category (nonterminal)
  - eg one for expressions, another for commands
- If Natural semantics used, tells how to compute final value from code
- If Transition semantics used, tells how to compute next "state"
  - To get final value, put in a loop

Natural Semantics Interpreter Implementation

- Identifiers: $(k,m) \Downarrow m(k)$
- Numerals are values: $(N,m) \Downarrow N$
- Conditionals: $\dfrac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$ $\dfrac{(B,m) \Downarrow \text{false} \quad (C',m) \Downarrow m'}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \Downarrow m'}$

```
compute_exp (Var(v), m) = look_up v m
compute_exp (Int(n), _) = Num (n)
…
compute_com (IfExp(b,c1,c2), m) =
       if compute_exp (b,m) = Bool(true)
       then compute_com (c1,m)
       else compute_com (c2,m)
```


Natural Semantics Interpreter Implementation

- Loop: $\dfrac{(B, m) \Downarrow \text{false}}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m}$ $\dfrac{(B,m) \Downarrow \text{true} \quad (C,m) \Downarrow m' \quad (\text{while } B \text{ do } C \text{ od}, m') \Downarrow m''}{(\text{while } B \text{ do } C \text{ od}, m) \Downarrow m''}$

```
compute_com (While(b,c), m) =
      if compute_exp (b,m) = Bool(false)
      then m
      else compute_com
              (While(b,c), compute_com(c,m))
```

- May fail to terminate - exceed stack limits
  - Returns no useful information then

Transition Semantics ("Small-step Semantics")

- Form of operational semantics
- **Describes how each program construct transforms machine state by *transitions***
- Rules look like

  (C, *m*) --> (C', *m'*) or (C,*m*) --> *m'*
- *C, C'* is code remaining to be executed
- m, m' represent the state/store/memory/environment
  - Partial mapping from identifiers to values
  - Sometimes *m* (or *C*) not needed
- Indicates exactly one step of computation

Expressions and Values

- *C, C'* used for commands; *E, E' for* expressions; *U, V* for values
- Special class of expressions designated as *values*
  - Eg 2, 3 are values, but 2+3 is only an expression
- Memory only holds values
  - Other possibilities exist

Evaluation Semantics

- Transitions successfully stops when E/*C* is a value/memory
- Evaluation fails if no transition possible, but not at value/memory
- Value/memory is the final *meaning* of original expression/command (in the given state)
- Coarse semantics: final value / memory
- More fine grained: whole transition sequence

Simple Imperative Programming Language

- I ∈ Identifiers
- N ∈ Numerals
- B ::= true | false | B & B | B or B | not B | E < E | E = E
- E::= N | I | E + E | E * E | E - E | - E
- C::= skip | C;C | I ::= E
  | if B then C else C fi | while B do C od

Transitions for Expressions

- Numerals are values
- Boolean values = {true, false}
- Identifiers: (k,m) --> (m(k), m)

Boolean Operations:

- Operators: (short-circuit)

(false & B, m) --> (false,m)
(true & B, m) --> (B,m)

$$\frac{(B, m) \dashrightarrow (B'', m)}{(B \,\&\, B', m) \dashrightarrow (B'' \,\&\, B', m)}$$

(true or B, m) --> (true,m)
(false or B, m) --> (B,m)

$$\frac{(B, m) \dashrightarrow (B'', m)}{(B \text{ or } B', m) \dashrightarrow (B'' \text{ or } B', m)}$$

(not true, m) --> (false,m)
(not false, m) --> (true,m)

$$\frac{(B, m) \dashrightarrow (B', m)}{(\text{not } B, m) \dashrightarrow (\text{not } B', m)}$$

Relations

$$\frac{(E, m) \dashrightarrow (E'', m)}{(E \sim E', m) \dashrightarrow (E'' \sim E', m)}$$

$$\frac{(E, m) \dashrightarrow (E', m)}{(V \sim E, m) \dashrightarrow (V \sim E', m)}$$

(U ~ V, m) --> (true,m) or (false,m)
depending on whether U ~ V holds or not

Arithmetic Expressions

$$\frac{(E, m) \dashrightarrow (E'', m)}{(E \text{ op } E', m) \dashrightarrow (E'' \text{ op } E', m)}$$

$$\frac{(E, m) \dashrightarrow (E', m)}{(V \text{ op } E, m) \dashrightarrow (V \text{ op } E', m)}$$

(U op V, m) --> (N,m)

where N is the specified value for U op V

Commands - in English

- ***skip*** means we're done evaluating
- When evaluating an ***assignment***, evaluate the expression first
- If the ***expression being assigned is already a value***, update the memory with the new value for the identifier
- When evaluating a ***sequence***, work on the first command in the sequence first
- If the first command evaluates to a new memory (i.e. it completes), evaluate remainder with the new memory

Commands

(skip, m) --> m

$$\frac{(E,m) \dashrightarrow (E',m)}{(k{:=}E,m) \dashrightarrow (k{:=}E',m)}$$

(k:=V,m) --> m[k <-- V]

$$\frac{(C,m) \dashrightarrow (C'',m')}{(C;C', m) \dashrightarrow (C'';C',m')} \qquad \frac{(C,m) \dashrightarrow m'}{(C;C', m) \dashrightarrow (C',m')}$$

If Then Else Command - in English

- If the boolean guard in an if_then_else is true, then evaluate the first branch
- If it is false, evaluate the second branch
- If the boolean guard is not a value, then start by evaluating it first.

If Then Else Command

- Base Cases:

  (if true then C else C' fi, m) --> *(C, m)*

  (if false then C else C' fi, m) --> *(C', m)*

- Recursive Case:

$$\frac{(B,m) \dashrightarrow (B',m)}{(\text{if } B \text{ then } C \text{ else } C' \text{ fi}, m) \dashrightarrow (\text{if } B' \text{ then } C \text{ else } C' \text{ fi}, m)}$$

While Command

(while B do C od, m) -->
(if B then C ; while B do C od else skip fi, m)

In English: Expand a While into a test of the boolean guard, with the true case being to do the body and then try the while loop again, and the false case being to stop.

Example Evaluation

- First step:

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> ?

Example Evaluation

- First step:

(x > 5, {x -> 7}) --> ?

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> ?

Example Evaluation

- First step:

(x,{x -> 7}) --> (7, {x -> 7})

(x > 5, {x -> 7}) --> ?

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> ?

Example Evaluation

- First step:

(x,{x -> 7}) --> (7, {x -> 7})

(x > 5, {x -> 7}) --> (7 > 5, {x -> 7})

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> ?

Example Evaluation

- First step:

(x,{x -> 7}) --> (7, {x -> 7})

(x > 5, {x -> 7}) --> (7 > 5, {x -> 7})

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> (if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi, {x -> 7})

Example Evaluation

- Second Step:

(7 > 5, {x -> 7}) --> (true, {x -> 7})

(if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})
--> (if true then y:=2 + 3 else y:=3 + 4 fi,
{x -> 7})

- Third Step:

(if true then y:=2 + 3 else y:=3 + 4 fi, {x -> 7})
-->(y:=2+3, {x->7})

Example Evaluation

- Fourth Step:

$$\frac{(2+3, \{x\text{-}> 7\}) \text{ --> } (5, \{x \text{-}> 7\})}{(y:=2+3, \{x\text{-}>7\}) \text{ --> } (y:=5, \{x\text{-}>7\})}$$

- Fifth Step:

  (y:=5, {x->7}) --> {y -> 5, x -> 7}

Example Evaluation

- Bottom Line:

(if x > 5 then y:= 2 + 3 else y:=3 + 4 fi, {x -> 7})
--> (if 7 > 5 then y:=2 + 3 else y:=3 + 4 fi, {x -> 7})
-->(if true then y:=2 + 3 else y:=3 + 4 fi, {x -> 7})
-->(y:=2+3, {x -> 7})
--> (y:=5, {x -> 7})
--> **{y -> 5, x -> 7}**

Adding Local Declarations

- Add to expressions:
- *E* ::= … | let *x* = *E* in *E'* | fun *x* -> *E* | *E E'*
- fun *x* -> *E* is a value
- Could handle local binding using state, but have assumption that evaluating expressions does not alter the environment
- We will use **substitution** here instead
- **Notation**: *E* [*E'* / *x*] means replace all free occurrence of *x* by *E'* in *E*

Calling Conventions (Common Strategies)

- Call by value: First evaluate the argument, then use its value
- Call by name: Refer to the computation by its name; evaluate every time it is called
- Call by need (lazy evaluation): Refer to the computation by its name, but once evaluated, store ("memoize") the result for future reuse

Call-by-value (Eager Evaluation)

(let k = V in E, m) --> (E [V/ k],m)

$$\frac{(E, m) \text{ --> } (E'',m)}{(\text{let } k = E \text{ in } E', m) \text{ --> } (\text{let } k = E'' \text{ in } E')}$$

((fun k -> E) V, m) --> (E [V / k], m)

$$\frac{(E', m) \text{ --> } (E'',m)}{((\text{fun } k \text{ -> } E)\ E', m) \text{ --> } ((\text{fun } k \text{ -> } E)\ E'', m)}$$

Call-by-name

- (let k = E in E', m) --> (E' [E / k], m)
- ((fun k -> E') E, m) --> (E' [E / k], m)
- Question: Does it make a difference?
- It can depending on the language

Transition Semantics Evaluation

- **A sequence of transitions**: trees of justification for each step

(C_1,m_1) --> (C_2,m_2) --> (C_3,m_3) --> … --> m

- **Definition:** let **-->*** be the transitive closure of **-->** i.e., the smallest transitive relation containing **-->**

Church-Rosser Property

- Church-Rosser Property: If E-->* E_1 and E-->* E_2, if there exists a value V such that E_1 -->* V, then E_2 -->* V
- Also called **confluence** or **diamond property**
- Example: (consider + as a function

E= 2 + 3 + 4

E_1 = 5 + 4 E_2= 2 + 7

V =9

Does Church-Rosser Property always Hold?

- No. Languages with side-effects tend not be Church-Rosser with the combination of call-by-name and call-by-value
- Benefit of Church-Rosser: can check equality of terms by evaluating them (but particular evaluation strategy might not always terminate!)
- Alonzo Church and Barkley Rosser proved in 1936 the λ-calculus does have it
  - λ-calculus → Coming up next!

Major Phases of a PicoML Interpreter